Der LZ77 Algorithmus

von

Stefan Mühlinghaus
Matrikelnummer: 11033988
Benutzerkennung: ai495

Über den Algorithmus:

Der LZ77 Algorithmus wurde von seinen Erfindern Abraham Lempel und Jacob Ziv erstmalig unter dem Namen “A Universal Algorithm for Sequential Data Compression” in der Zeitschrift “IEEE Transactions on Information Theory”, Ausgabe 23 vom 3. Mai 1977 (daher der Name, “Lempel-Ziv-1977”) veröffentlicht. Sie besrieben darin einen anderen Weg der Datenkompression, die bis dahin nur durch profunde Kenntnis der zu komprimierenden Daten möglich war, oder sich ausschliesslich auf eine Entropie-Codierung beschränkte. Mit dem LZ77 Algorithmus war es jetzt möglich, beliebige Daten zu komprimieren, ohne vorherige Kenntnis ihrer Struktur oder Grösse. Erzielt wird dies, indem in den Eingabedaten redundante Muster erkannt und durch Verweise auf vorherige Vorkommen ersetzt werden. Folglich handelt es sich, logischerweise, um eine verlustfreie Art der Kompression. Bereits ein Jahr später, 1978, stellten Lempel und Ziv einen weiteren Algorithmus (LZ78) vor, der die Daten mit Hilfe eines selbst erstellten, und beim Dekomprimieren reproduzierbaren, Wörterbuches komprimiert. Dieser fand jedoch, obgleich technisch überlegener, keine so grosse Annahme, da er im Gegensatz zu LZ77, der “freies Gedankengut” ist, unter unvorteilhaften Copyright-Richtlinien steht.

Funktionsweise:

Bei LZ77 handelt es sich um einen so-genannten Gleitfenster (“Sliding Window”) Algorithmus. Das liegt in der Tatsache begründet, dass er immer nur einen kleinen Abschnitt (Lookahead-Fenster) der Eingabedaten gleichzeitig “sieht” und diesen komprimiert. Danach bewegt sich das Fenster um die Anzahl der gerade komprimierten Zeichen weiter. Darüber hinaus bedient sich LZ77 eines weiteren Gleitfensters, des Datenfensters, das parallel mit dem Lookahead-Fenster über die Daten wandert und die zuletzt komprimierten Zeichen enthält. In diesem Fenster wird bei jedem Schritt das erste Vorkommen der grössten Teilmenge der im Lookahead-Fenster befindlichen Zeichen gesucht, und falls vorhanden mit deren Index im Datenfenster und deren Länge codiert. Wird nichts gefunden, so wird dies als (0,0) kodiert. Anschliessend wird diesem Tupel noch das erste nicht übereinstimmende Zeichen angefügt und dann die beiden Gleitfenster um die Länge der so codierten Zeichen verschoben. Falls eine Übereinstimmung *aller* im Lookahead-Fenster befindlichen Zeichen gefunden wird, werden diese aber trotzdem *nicht* alle codiert, sondern nur alle bis auf das letzte, welches dann als nicht übereinstimmendes Zeichen angehängt wird. Dies ist notwendig, um die Form der komprimierten Daten zu wahren.

Beispiel: Codierung des Wortes “abracadabra”. Die Fenstergrössen stehen in Klammern. Gefundene Übereinstimmungen sind grün, die erste Nicht-Übereinstimmung ist rot.

Datenfenster(10)	Lookahead(4)	Restdaten	Codierung
		abracadabra	
	abra	cadabra	(0,0)a
a	brac	adabra	(0,0)b
ab	raca	dabra	(0,0)r
abr	acad	abra	(1,1)c
abrac	adab	ra	(1,1)d
abracad	abra		(1,4)? → Fehler!
abracad	abra		(1,3)a
abracadabra	Leer → Ende		

Bei der rot hervorgehobenen Zeile handelt es sich um eine mögliche Kodierung aller Daten im Lookahead-Fenster, die jedoch falsch wäre (s.o.).

Nachteile der LZ77 Kompression:

Da sich der LZ77 Algorithmus keine Informationen über die zu codieren Daten zunutze machen kann, operiert er auf verschiedenen Eingabedaten unterschiedlich gut oder schlecht.

Sollte er beispielsweise ein Bild komprimieren, würde ihm das recht leidlich gelingen. Handelt es sich bei diesem Bild jedoch um einen vertikalen Farbverlauf, in dem die Pixel einer Zeile jeweils den gleichen Farbwert haben, so müsste LZ77 dies erst “erkennen”, indem er zuerst das erste, dann die nächsten beiden, dann die nächsten 4, usw. Pixel komprimiert bis die Zeile abgearbeitet ist. Bei der nächsten Zeile müsste er komplett neu beginnen, da die Farbwerte aller Zeilen sich unterscheiden. Eine einigermassen effektive Komprimierung wenn man nichts über die Natur der Daten weiss.

Kennt man sie allerdings und weiss, dass es sich um einen vertikalen Farbverlauf handelt, so könnte man effektiver einfach die Breite des Bildes einmalig und für jede Zeile ihren Farbwert speichern. Dies stellt einen erheblichen Vorteil der LZ77 Methode dar. Und sie kann sogar noch verbessert werden. Kennt man nämlich den Algorithmus, mit dem der Farbverlauf erzeugt wurde, so kann man ein Bild beliebiger Höhe und Breite einzig mit diesen beiden Werten sowie dem Anfangs- und End-Farbwert darstellen. Beim Dekomprimieren könnte daraus der ursprüngliche Farbverlauf fehlerlos rekonstruiert werden.

Auch ist LZ77 nur in der Lage, Redundanzen innerhalb eines vergleichsweise kleinen Abschnitts der Eingabedaten zu erkennen und zu substituieren. Aus diesem Grund wurden bis zum heutigen Tag die verschiedensten Änderungen am Algorithmus durchgeführt, sowie Anpassungen für eine Vielzahl von Spezialfällen. Aber auch trotz dieser Bemühungen gilt nach wie vor: *Kein einzelner Algorithmus komprimiert jede Art von Daten gleich gut!*

Asymptotische Laufzeitanalysen:

l = Länge des Lookahead-Fensters
d = Länge des Datenfensters
n = Menge der Daten

Komprimieren:

Worst Case: $l + (n - l) * (l - 1) * (d - l - 1) = l^3 - nl^2 - dl^2 + nld - nd + dl + n \Rightarrow \mathbf{\Theta(n)}$

Best Case: $(n - l) / l + l \Rightarrow \mathbf{\Theta(n)}$

Dekomprimieren:

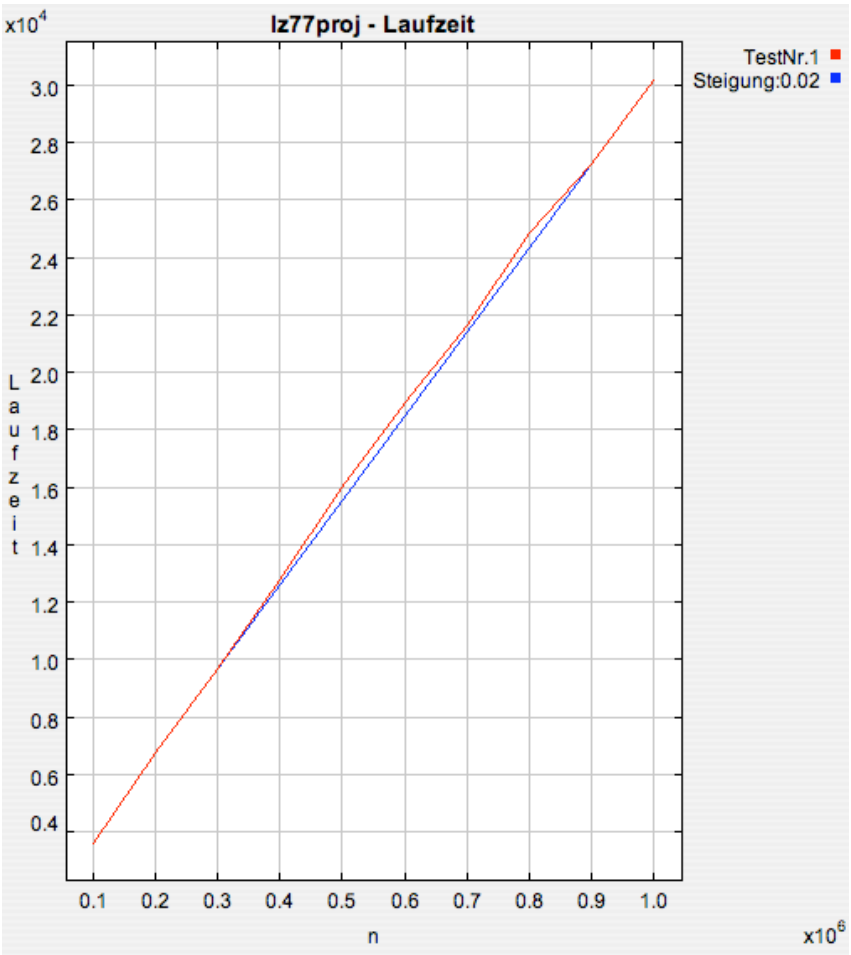
Worst Case: $n \Rightarrow \mathbf{\Theta(n)}$

Best Case: $n \Rightarrow \mathbf{\Theta(n)}$

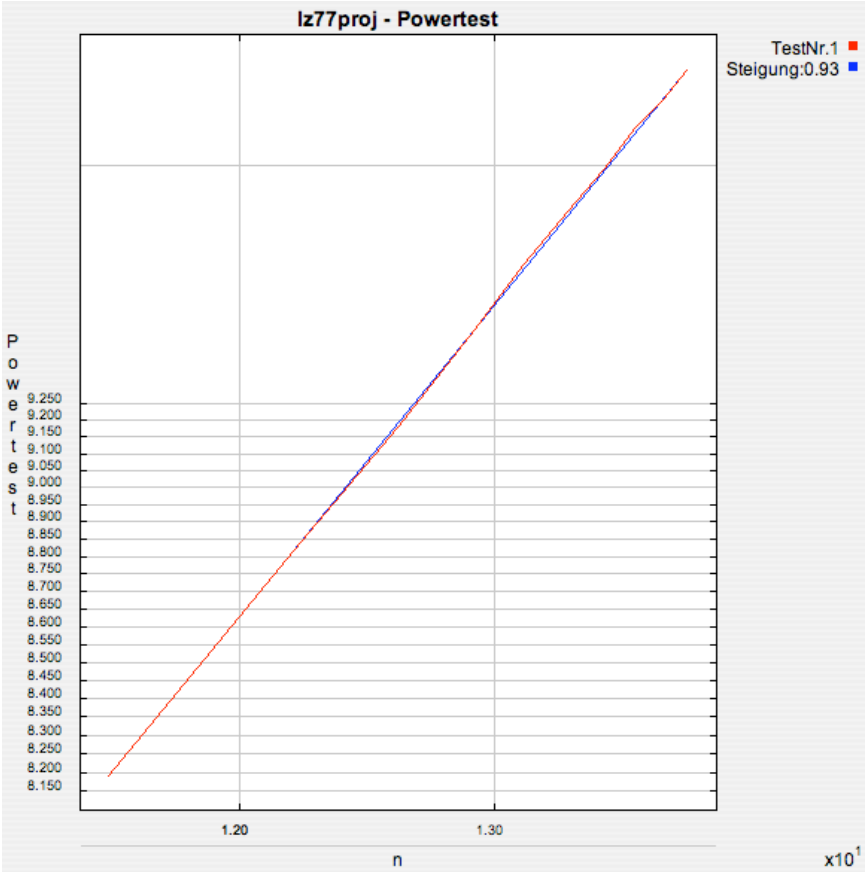
Experimentelle Laufzeitanalysen:

Menge der Daten = 100.000 - 1.000.000, Schrittweite 100.000
Länge des Datenfensters = 60
Länge des Lookahead-Fensters = 60

Komprimieren:

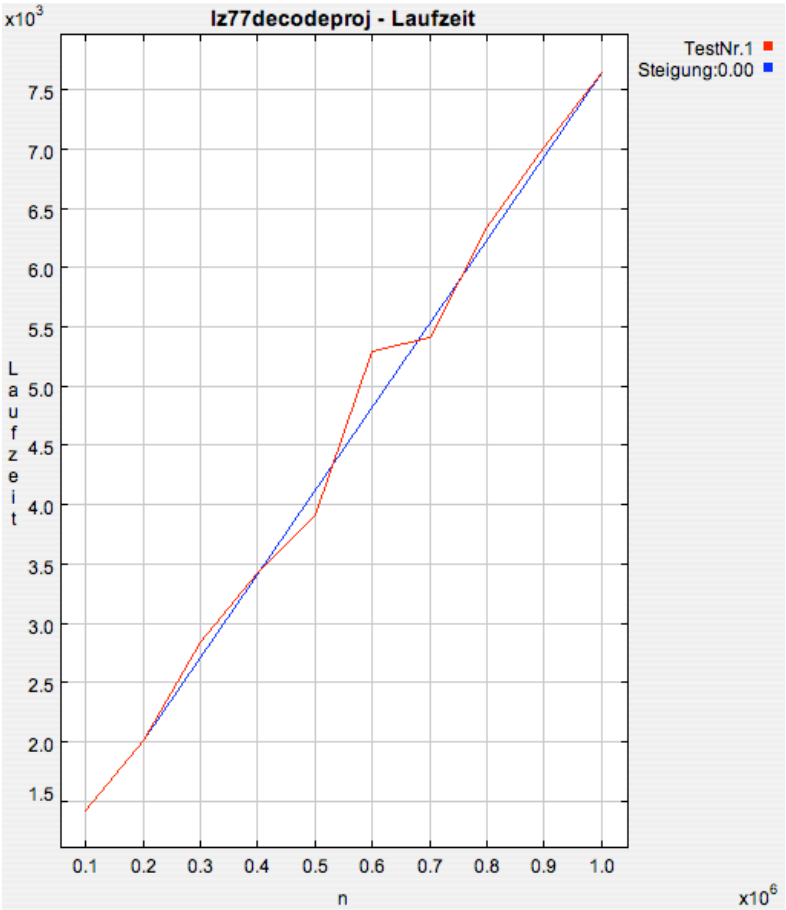


Bei der Laufzeitanalyse kann man deutlich erkennen, dass es sich bei dem vom LZ77 Algorithmus erzeugten Graphen (Rot) um eine linear ansteigende Gerade handelt. Dies belegt, dass es sich um ein Laufzeitverhalten von **Θ(n)** handeln muss. Zum Vergleich wurde aus zwei Punkten des Graphen eine weite Gerade (Blau) erstellt, die mit der ursprünglichen stark übereinstimmt und eine Steigung von **0.02** besitzt.

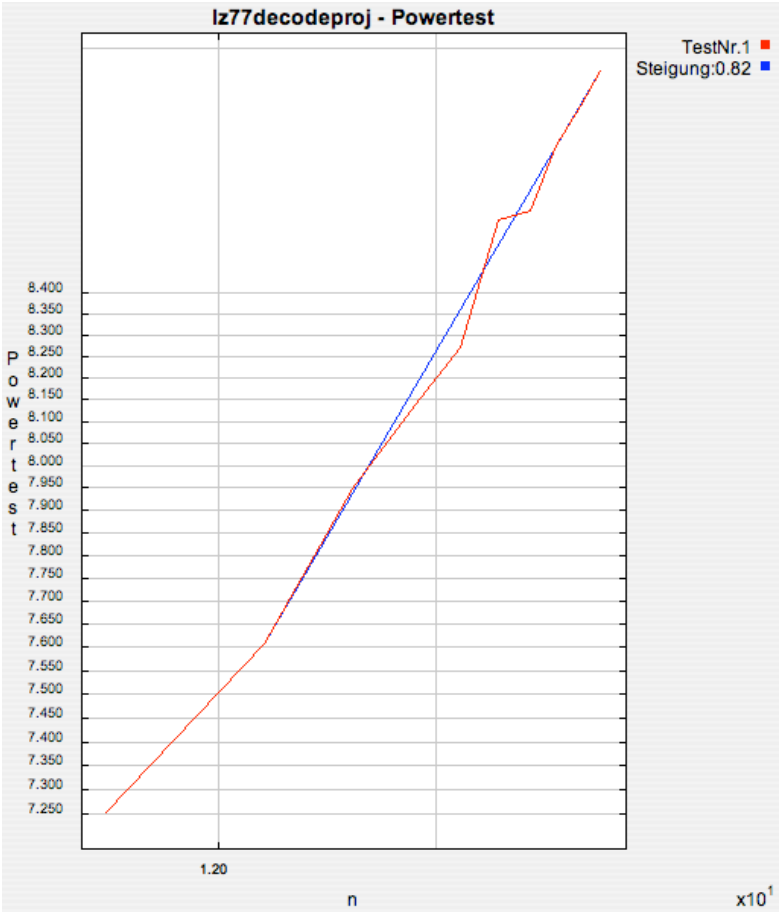


Auch beim Powertest auf logarithmischer Skala ergibt sich ungefähr eine Gerade (Rot). Wieder wurde zum Vergleich eine stark übereinstimmende Gerade (Blau) aus zwei Punkten der Ursprungsgraden berechnet. Diese besitzt eine Steigung von **0.93**, was fast genau **1** entspricht, wodurch wiederum ein Laufzeitverhalten von **Θ(n)** belegt wird.

Dekomprimieren:



Da beim Dekomprimieren für jedes Eingabetupel höchstens eine Kopieraktion der indizierten Werte durchgeführt werden muss, ist auch hier die Laufzeit offensichtlich **Θ(n)**. Die Laufzeitanalyse belegt dies, da es sich bei ihrem Graphen, von einigen “Ausreißern” abgesehen, um eine Gerade handelt.



Auch anhand des Powertests lässt sich diese Behauptung untermauern, da die angenäherte Steigung von **0.82**, wenn überhaupt, zu **Θ(n)** gerechnet werden muss.

<u>Die wichtigsten Weiterentwicklungen:</u>

Lempel-Ziv-1978 (LZ78)

Bereits ein Jahr nach dem LZ77 Algorithmus, veröffentlichten Lempel und Ziv einen weiteren verlustlosen Kompressionsalgorithmus unter dem Namen “Compression of Individual Sequences via Variable-Rate Coding”. Dieser wurde im folgenden unter dem Namen LZ78 bekannt. Der Algorithmus setzt auf LZ77 auf, erweitert ihn aber in so fern, als dass nun ein Wörterbuch der bisher kodierten Zeichenketten zur Laufzeit erstellt wird und statt der Kombination Index + Länge wird nun ein Index auf dieses Wörterbuch kodiert wird. Da das Wörterbuch aus praktischen Gesichtspunkten natürlich nicht beliebig gross werden darf, ist an dieser Stelle zusätzlicher Aufwand zur Pflege erforderlich, was den Algorithmus verlangsamt. LZ78 wird in der “Praxis” eher ungern benutzt, da es gewissen Copyright Richtlinien unterliegt und sich deshalb nicht gut für “freie” Software eignet.

Lempel-Ziv-Storer-Szymanski (LZSS)

1982 wurde von den Autoren James A. Storer und Tomas G. Szymanski ein Artikel namens “Data Compression via Textual Substitution” veröffentlicht, in dem sie einen weiteren, auf LZ77 aufsetzenden Algorithmus beschrieben, später bekannt als LZSS. Die Hauptunterschiede zu LZ77 bestehen darin, dass er ähnlich wie LZ78 ein Wörterbuch der bisher Kodierten Zeichen führt und dass er nur Sequenzen kodiert, die eine bestimmte Mindestlänge erreichen. Letzteres bedeutet logischerweise, dass er nicht jedes Zeichen bzw. jede Sequenz komprimiert. Dies ist möglich, indem er nicht komprimierte Zeichen einfach zusammen mit den Komprimierten Indizes in einem String ausgibt. An dieser Stelle ist dann eine wie auch immer geartete Overhead-Markierung nötig, um das eine von dem anderen zu unterscheiden. Die Komprimierung besteht nicht wie bei LZ78 aus einem Index auf das Wörterbuch, sondern wie von LZ77 bekannt aus der Kombination Datenfensterindex + Länge, die aus dem Wörterbuch geholt werden. Das Anfügen des ersten nicht übereinstimmenden Zeichens entfällt, da nicht komprimierte Zeichen auch so in den Ausgabestring geschrieben werden.

Lempel-Ziv-Welch (LZW)

Terry A. Welch veröffentlichte 1984 den Artikel “A Technique for High-Performance Data Compression”, in dem er ein neues, auf LZ78 aufsetzendes Kompressionsverfahren beschrieb. Dieser Algorithmus produziert als Ausgabe *ausschliesslich* Indizes auf das Wörterbuch. Dies wird möglich, indem die ersten 256 Einträge des Wörterbuchs mit allen einzelnen Zeichen vorbesetzt werden, auf die der Algorithmus also schon beim kodieren der ersten Eingabesequenz zurückgreifen kann. Wie auch schon LZ78 ist LZW (logischerweise) nicht frei benutzbar und wird deshalb eher ungern benutzt. Trotzdem gibt es eine Reihe von Anwendungsfällen, in denen LZW in der Praxis genutzt wird. Unter anderem die Gafikformate GIF und TIF, wobei letzteres auch einige andere Kompressionsverfahren unterstützt (z.B. Deflate).

Deflate

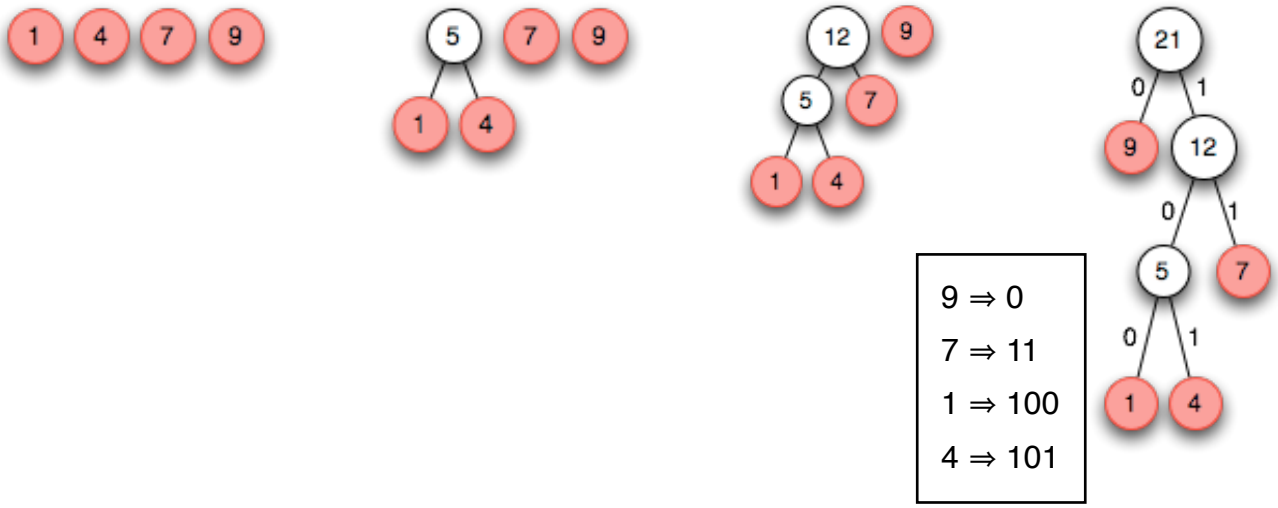
Deflate ist der heutzutage wohl am weitesten verbreitete Kompressionsalgorithmus, was wohl darauf zurückzuführen ist, dass er mit keinerlei Copyright-Richtlinien oder Patenten behaftet ist. Es handelt sich dabei grundlegend um den LZSS Algorithmus mit anschliessender Entropiekodierung (Huffmann). Das Deflate Verfahren wurde erstmals von Phillip W. Katz für die von ihm erfundene ZIP-Kompression eingesetzt. Seine Spezifikation wurde 1996 im RFC-1951 festgehalten. Es findet in vielen bekannten Kompressions- und Grafikformaten Anwendung, wie z.B. ZIP, PNG, TIFF oder PDF.

Verbesserung: Entropie Kodierung

Da nach einem Durchlauf von LZ77 weitestgehend alle Redundanzen aus den Eingabedaten entfernt wurden, kann durch einen weiteren Durchlauf keine weitere Komprimierung der Ausgabedaten erzielt werden. Sogar das Gegenteil wäre der Fall, da jedem nicht-redundanten Byte, in diesem Fall fast allen, zwei Werte für Index und Länge angehängt werden. Im Fall von Short-Werten würden die Ausgabedaten beinahe die fünffache Größe der Eingabedaten haben. Können also die Ausgabedaten noch weiter komprimiert werden? Ja, das können sie. Auf dem gleichen Weg, den auch der Deflate-Algorithmus beschreitet: Mit einer Entropie-Kodierung, beispielsweise nach Huffman.

Die Idee dahinter beruht auf der Tatsache, dass verschiedene Zeichen in den Eingabedaten unterschiedlich oft vorkommen. Wenn die öfter vorkommenden Zeichen dann mit einem kürzeren Codewert codiert werden, kann man auf diese Weise eine Verkleinerung der Ausgabedaten erreichen.

Der von David Huffman entwickelte Algorithmus weist jedem Zeichen seine Häufigkeit in den Eingabedaten zu. Danach fasst er immer die beiden Zeichen mit den niedrigsten Werten als Kinderknoten unter einem Dummyknoten zusammen. Die Häufigkeit dieses Vaterknotens wird die Summe der Häufigkeiten der beiden Kinder. Mit diesem Verfahren wird so lange fortgefahren, bis alle Zeichen in einem einzigen binären Baum mit einem Dummy-Element als Wurzel angeordnet sind. In diesem Baum werden dann die linken Kannten mit 0 und die rechten mit 1 betitelt. Die Kodierung der einzelnen Zeichen ergibt sich dann aus der Reihenfolge der Kannten, die den Weg zu ihrem Knoten ausmachen.



Implementierung

ByteElement

Die Klasse ByteElement stellt ein einzelnes Byte in einer einfach verketteten Liste dar. Auf diese Weise werden die Einträge in den Gleitfenstern gespeichert.

ByteElementPool

Diese Klasse dient vor allem dem Speichermanagement. Von ihr können die Gleitfenster ByteElement-Objekte erhalten und bei ihr können sie solche wieder abgeben, wenn sie nicht mehr benötigt werden. Dadurch wird nicht nur der Speicherverbrauch begrenzt, sondern auch eine Garbage Collection unnötig gemacht. Vor allem spart man sich jedoch im weiteren Verlauf des Programms den Overhead von immer wieder neu instanzierten Objekten.

Huffman

Hierbei handelt es sich um eine Wrapper-Klasse um die beiden Klassen “Deflater” und “Inflater”. Die Komprimierung wird mit dem Parameter “HUFFMAN_ONLY” aufgerufen, um die Anwendung weiterer Algorithmen zu unterdrücken.

LZ77

Kernklasse der Programmlogik, mit ihr kann ein Eingabestream nach dem LZ77-Algorithmus in einen Ausgabestream komprimiert werden. Diese Implementation beruht auf der Anwendung der SlidingByteWindow-Klasse und den damit zusammen hängenden Klassen. Eine ältere Version die auf Arrays basiert kann unter “LZ77_alt.java” eingesehen werden.

SlidingByteWindow

Diese Klasse beinhaltet die Implementierung der Gleitfenster auf Basis einer einfach verketteten Liste. Da die Standard Java Klassen gewisse, für die Laufzeit sehr kritische, Operationen nicht oder nur zu umständlich beherrschten, ist dies eine völlig neue Klasse. Sie bietet eine **erhebliche** Performanceverbesserung gegenüber der Variante mit Arrays.

Benutzeroberflächen

Test

Die Klasse Test stellt ein einfaches Kommandozeileninterface zu LZ77 und Huffman dar. Sie nimmt Eingabe- und Ausgabedatei sowie einen auszuführenden Befehl und ggf. (im Fall von LZ77) noch weitere Parameter für die Fenstergrößen an. Mögliche Parameter sind:

```
encode <Eingabedatei> <Ausgabedatei> <Datenfenster> <Lookahead-Fenster>
decode <Eingabedatei> <Ausgabedatei> <Datenfenster>
huffman <Eingabedatei> <Ausgabedatei>
dehuffman <Eingabedatei> <Ausgabedatei>
```

Simple_LZ77_Compressor.jar

Hierbei handelt es sich um eine mit NetBeans erstellte GUI. Sie erlaubt die Eingabe von Eingabe- und Ausgabedatei, die Wahl einer Operation (Komprimieren oder Dekomprimieren) und die Angabe von Fenstergrößen für Datenfenster und Lookahead-Fenster. Mit einem Klick auf den “Los!”-Knopf wird die ausgewählte Aktion durchgeführt. Im Hintergrund wird nach der LZ77-Kompression automatisch noch eine Huffman-codierung durchgeführt.